Secondary vertex Finders documentation

`StRoot/StSecondaryVertexMaker/ :`

```
StV0FinderMaker.cxx
StV0FinderMaker.h
StXiFinderMaker.cxx
StXiFinderMaker.h
StKinkMaker.cxx
StKinkMaker.h
StKinkLocalTrack.cc
StKinkLocalTrack.hh
```

`StRoot/StSecondaryVertexMaker/doc/ :`

```
docXiFinder.tex
```

– **Strangeness group** –

People to contact :

Betty Bezverkhny	—	SVT
Gene van Buren	—	V0Finder
Helen Caines	—	SVT
Julien Faivre	—	V0Finder and XiFinder
Camelia Mironov	—	KinkFinder

(Up-to-date e-mail adresses can be taken from the hypernews).

Contents

List of Figures

1 What are the V0Finder and XiFinder ?

Let's first define the 3 different sorts of secondary vertices :

- Kink vertices : when a charged particle decays into a charged plus a neutral,
- V0 vertices : when a neutral particle decays into two charged particles,
- Xi vertices : when a charged particle decays into a charged plus a neutral, and then the neutral daughter decays itself into two charged particles. One can't say that a Xi vertex is a Kink followed by a V0, because both charged tracks (mother and daughter) have to be seen in the TPC in order to say that the vertex is a kink. This is not the case for Xi vertices, because the $c\tau$ of particles that do a Xi vertex is much shorter than the distance between the primary vertex and the TPC (50 cm), even shorter than the distance to the first layer of the SVT (6.7 cm, versus $c\tau_{\Xi} = 4.9$ cm and $c\tau_{\Omega} = 2.5$ cm).

People working on strange particles in the strangeness group need a piece of code that is able to reconstruct the primary strange particles from the tracks. Those strange particles can be divided into two groups :

- **V0's :** those particles (Λ, K^0_s) decay into two particles ($\Lambda \longrightarrow p\pi^-$, $K^0_s \longrightarrow \pi^+\pi^-$). We need to be able to search and find them using only the daughters' tracks (that's all we have !).
- **Xi's :** those particles (Ξ, Ω) decay into 1 charged particle – called bachelor – and a Λ ($\Xi \longrightarrow \Lambda\pi^-$, $\Omega \longrightarrow \Lambda K^-$). The Λ, as said in the previous item, decays into two particles. So here, we need to find all combinations of 3 charged particles that are possibly the daughters of a unique strange particle.

The algorithms of those codes are described in section 3.

From the beginning of STAR and until year 2002, the codes used were what will be called in this documentation *exiam* and *ev0am*. They are Fortran codes, located in `pams/global/exi/exiam.F` and `pams/global/ev0/ev0_am2.F`, with other files that are necessary for the code to be run (basically subroutines and interfacing functions).

So here is the "old" way to do things : the BFC is run, and it calls the series of makers that it is supposed to call. Among those makers are `StXiMaker` and `StV0Maker` – the strangeness makers, with `StKinkMaker` – and they are run after nearly all the other makers of the BFC, since they need the tracks to be reconstructed and the primary vertex to be found. The V0Maker is run first, finds the V0's, and those feed the XiMaker, which tries to find Xi candidates for each V0. At a deeper level, inside the `Make()` function of both the XiMaker and the V0Maker, are called the Fortran PAMs, i.e. respectively `ev0am` and `exiam`. PAM means either "Plugable Analysis Module" or "Physics Analysis Module". I don't know if somebody knows which of the two it is !

The interfacing between the BFC (C++) and the PAMs (Fortran) won't be discussed here. If you want to know more, you can have a look at those files : `pams/global/exi/exiam.idl`, `pams/global/-ev0/ev0_am2.idl`, `pams/idl/dst_track.idl`, `pams/idl/dst_vertex.idl`, `pams/idl/dst_v0_vertex.-idl`, `pams/idl/dst_xi_vertex.idl`. The data are passed – from maker to maker, and also between a maker and the PAM that it calls – by tables. For example the tracks are stored in a table, and ev0am will read the table to have the tracks' parameters. It will then store the V0's in another table. Then, exiam will read this table and the table of tracks, and write the Xi's found in a third table. Classes for interfacing have a general name which is "St_tableName_Table.h", and they can be found in `include/tables/` (as examples : `St_exi_exipar_Table.h`, `St_ev0_aux_Table.h`, `St_dst_track_Table.h`, `St_dst_xi_vertex_Table.h`, etc... Further information about St_dst_track_Table can be found at `root.cern.ch/root/html/TTable.html` in the section "Class description"). On the other hand, the corresponding structures are stored in `pams/global/idl/`, in files like `exi_exipar.idl` (general name of the file : tableName.idl ; general name of the structure used afterwards : tableName_st). You could have a look at e.g. `StXiMaker::Init()` (the V0 and XiMaker's are in `StRoot/St_dst_Maker/`) to have an example of how all this is used.

As explained in section 2, the strangeness Fortran PAMs had to be replaced with C++ code. As exiam and ev0am refer to the corresponding PAMs, *XiFinder* and *V0Finder* are the names we gave to their C++ translations. They are makers, whose complete names are `StXiFinderMaker` and `StV0FinderMaker`. This is what we call the *strangeness StSecondaryVertexMaker package*.

2 Historical purpose for StSecondaryVertexMaker package

The StSecondaryVertexMaker package implements secondary vertex-finding in C++. *Why ?* There is and has been code for reconstucting secondary vertices in the form of the `ev0`, `exi`, and `tkf` PAMs. These PAMs were written in Fortran and work with tables. They are called from C++ makers currently kept in the St_dst_Maker package library. They work well, but suffer limitations :

- They cannot be re-run on DSTs after production,
- They cannot operate on the tracking output of ITTF.

In order to overcome both limitations, the preferred solution is to write C++ versions of these PAMs which can use StEvent structures for both input and output (versus trying to convert StEvent structures back into tables for input). This is the primary purpose of the StSecondaryVertexMaker package.

Advantages of being able to run such a code on the DSTs are that analysis such as rotating can be done without running the whole BFC on the daq files, as well as analysis that require modifications in the secondary vertex reconstruction code, like the value of the reconstruction cuts for example. The time profit is huge, since it takes more than 20 times more time to run the whole reconstruction chain than just the C++ secondary vertex reconstruction makers.

3 Overall algorithms of the codes

Apart from some parts described below, the V0Finder and XiFinder are essentially the ev0 and exi PAMs rewritten from Fortran to C++, from a "tabelized" way of communicating to a standard object-oriented, mono-language code.

3.1 KinkFinder

To be written.

3.2 V0Finder

Cut parameters are initially requested from the database (time stamps determine what is the nature of the data, e.g. `p-p`, `Au-Au`, `d-Au`). Then, tracks which satisfy a set of cuts are chosen as candidates for V0 daughters. The daughter candidates are then examined in pairs of negative and positive daughters to see if the tracks approach each other and pass a series of cuts to determine if they are consistent with a V0 secondary decay.

Formerly, a second pass was required on the V0s. This was to facilitate their use in finding Xi decays. V0s from Xi decays are *secondary* V0s, and thus do not originate from the primary vertex. This means looser cuts are necessary on these V0s than *primary* V0s. So, the first pass was made with the loose cuts, the Xi decays were found, and then the V0s were run through tighter cuts to remove unused ones which were inconsistent with being primary V0s. This second pass prevented the output of significant numbers of unnecessary V0 candidates.

This second pass has been replaced by a different mechanism. Now, for each V0 which passes the looser secondary V0 cuts, a `UseV0()` function is called. The idea is that a XiFinder can be written which inherits from the V0Finder, and implements the `UseV0()` function to find Xi candidates with a given V0. The `UseV0()` function then returns true or false depending on whether any Xi candidates are found using that V0. Upon returning to the V0Finder code, the V0 is discarded if it neither passes the cuts for a primary V0 nor gets used in the `UseV0()` function.

This scheme has the advantage of not inserting a V0 into the StEvent vector of V0s unless it is a viable candidate. It also reduces considerably the memory overhead required during V0-finding as not all of the secondary V0 candidates are found and stored at once (particularly poignant in high-multiplicity events where many thousands of secondary V0s are considered). This only disadvantage is some overhead in making a function call in the middle of the V0-finding loop.

FIG. 1 shows that even if the code of the V0Finder is quite long, the alorithm is definitely simple.

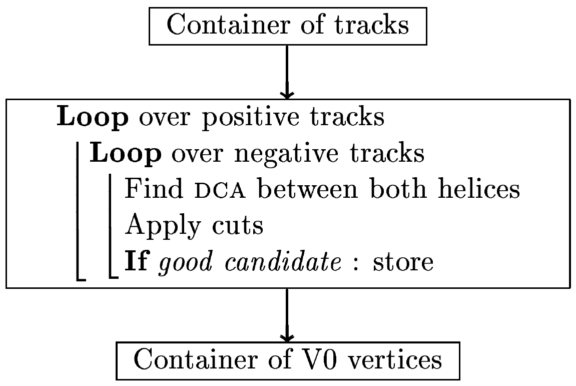

Container of tracks
Loop over positive tracks
Loop over negative tracks
Find DCA between both helices
Apply cuts
If *good candidate* : store
Container of V0 vertices

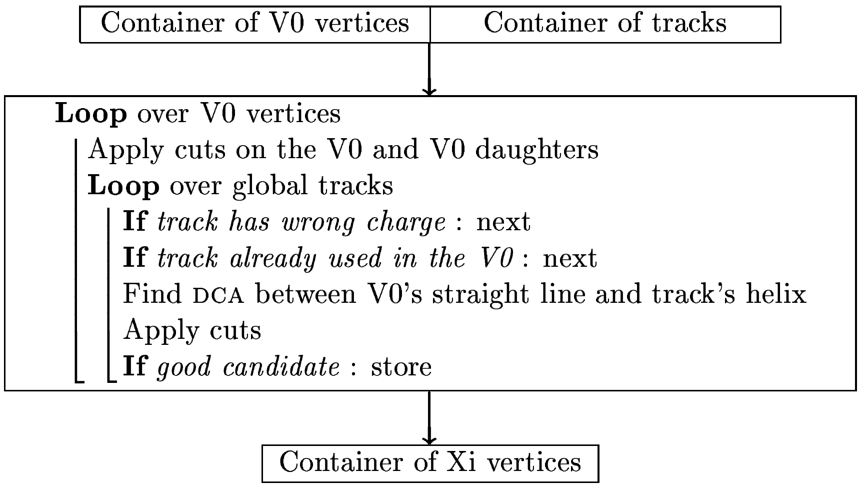
FIG. 1 : *Overall algorithm of the V0Finder.*

3.3 XiFinder

The `StXiFinderMaker` inherits from `StV0FinderMaker` as indicated above. Because it is actually a V0Finder itself via this inheritance, one need not instantiate a `StV0FinderMaker` if one instantiates a `StXiFinderMaker`.

Similar to the `StV0FinderMaker`, appropriate cut parameters are initially requested from the database. The same tracks that are considered for the V0s are also used as daughter candidates for the Xis. The `Make()` member function then simply calls the inherited `Make()` member function from the `StV0FinderMaker`. Control comes back to the `StXiFinderMaker` at the call to `UseV0()`, which is implemented here as a loop over Xi daughter candidates to be paired with the V0 daughter candidate. If the daughters approach each other and pass a series of cuts, the candidate is accepted and stored in StEvent. Control is then passed back to the V0Finder, with a return value indicating whether the V0 was in fact used.

As for the V0Finder, FIG. 2 below shows that the XiFinder alorithm is very simple, although the code is long. More detailed algorithms can be found in section 4.



```
Container of V0 vertices | Container of tracks

Loop over V0 vertices
  Apply cuts on the V0 and V0 daughters
  Loop over global tracks
    If track has wrong charge : next
    If track already used in the V0 : next
    Find DCA between V0's straight line and track's helix
    Apply cuts
    If good candidate : store

Container of Xi vertices
```

FIG. 2 : *Overall algorithm of the XiFinder.*

4 Structure of the Fortran and C++ codes

4.1 KinkFinder

To be written.

4.2 Common remarks for the V0/XiFinder

The first thing to mention is a change in the interaction between the V0Finder and the XiFinder. The XiFinder actually uses the V0's that have first been found by the V0Finder. The table below shows how this is done in the Fortran code :

Call `StV0Maker::Make()`	Finds V0's and store them in a table
Call `StXiMaker::Make()`	Loops over the V0's in the table to find Xi candidates
Call `StV0Maker::Trim()`	Loops over the V0's in the table to throw away the non-primary V0's that are not used in a Xi candidate

This waste of memory (storing V0's that will be deleted afterwards) and of time (scanning twice the table of V0's) is solved in the C++ code, by the fact that the class `StXiFinderMaker` actually inherits from `StV0FinderMaker`. The member-functions of each of them and their role are listed in FIG. 3.

Note that `StXiFinderMaker::Init()` is the equivalent of `StV0FinderMaker::GetPars()`. The effect of the function `StV0FinderMaker::DontZapV0s` is that the V0's that are already in StEvent are kept, the V0's found by the V0Finder will be added. The function `StV0FinderMaker::UseExistingV0s` also keeps the V0's that are already in StEvent, but also prevents the V0Finder to be run, and forces the XiFinder to use the V0's previously found. The function `StV0FinderMaker::UseITTFTracks` has been implemented for ITTF test purposes, since it allows the user to choose between using the ITTF tracks or using the tracks found by the reconstruction code used up to now (TPT).

When the V0Finder is run alone, the method `StV0FinderMaker::UseV0()` is called for each V0 found, and returns `false`, since we don't want to find Xi's.

When both the XiFinder and V0Finder are run, the XiFinder first calls `StV0FinderMaker::Make()`. This function finds V0's and, for each of them, calls the `UseV0()` method. The latter runs the XiFinder algorithm, that will eventually tell `StV0FinderMaker::Make()` if the current V0 has been used in Xi candidates or not. This way to do, compared with the Fortran one, saves time and memory.

Function	Role in the V0Finder	Role in the XiFinder
`Init`	Inits	Gets `exipar` from the database
`Make`	V0Finder "central" algorithm	Calls the V0Finder
`Clear`	Clears	Not redefined
`GetPars`	Gets `ev0par2` from the database	Not redefined (not used)
`Prepare`	Finds event-wise parameters, fills tables	Not redefined
`UseV0`	Returns `false`	XiFinder "central" algorithm
`UseExistingV0s`	Sets a boolean flag	Not redefined
`DontZapV0s`	Sets a boolean flag	Not redefined
`SetTrackerUsage`	Sets a integer flag	Not redefined
`GetTrackerUsage`	Returns a integer flag	Not redefined
`SetSVTUsage`	Sets a integer flag	Not redefined
`GetSVTUsage`	Returns a integer flag	Not redefined
`SetV0LanguageUsage`	Sets a integer flag	Not redefined
`GetV0LanguageUsage`	Returns a integer flag	Not redefined
`SetXiLanguageUsage`	Sets a integer flag	Not redefined
`GetXiLanguageUsage`	Returns a integer flag	Not redefined
`SetLanguageUsage`	Sets a integer flag	Not redefined
`GetLanguageUsage`	Returns a integer flag	Not redefined
`SetLikesignUsage`	Sets a integer flag	Not redefined
`GetLikesignUsage`	Returns a integer flag	Not redefined
`SetRotatingUsage`	Sets a integer flag	Not redefined
`GetRotatingUsage`	Returns a integer flag	Not redefined
`SetEventUsage`	Sets a integer flag	Not redefined
`GetEventUsage`	Returns a integer flag	Not redefined
`Trim`	Remove the V0's that don't pass cuts	Not redefined (not used)

FIG. 3 : *Member-functions of* `StV0FinderMaker` *and* `StXiFinderMaker`.

The table below shows how this is run in the C++ XiFinder (provided that it's the XiFinder that is called and not just the V0Finder) :

Call `StXiFinderMaker::Make()`	(Calls everything below)
Call `StXiFinderMaker::Prepare()`	Finds event-wise parameters and fills the tables
Call `StV0FinderMaker::Make()`	Finds V0's in this event
Call `StXiFinder::UseV0()`	Finds Xi's for a given V0 and store them in StEvent
(back in `V0FinderMaker::Make`)	If V0 is used in Xi's / may be primary : stores in StEvent

4.3 V0Finder

FIG. 4 shows the detailed structure of the V0Finder, with all the cuts that are applied. Of course, more accurate information can be found... by looking at the code ;-) .

```
Get parameters from database
Get event
Get position of the primary vertex
Loop over all tracks
  Select TPT vs ITTF
  If bad flag : next
  If bad detector ID : next
  If no geometry : next
  CUT 1 on number of hits
  If 1st track : get magnetic field
  Store track and parameters in tables
            (separately pos. and neg.)
Loop over positive tracks
  Loop over negative tracks
    Select TPT vs ITTF tracks
    Determine V0's detector ID
    CUT 2 on number of hits
    CUT 1 on dcaTrackToPvx if "low" $p_\perp$
    Find number of intersection points between both helices
    Find 2D dca between both helices at both intersection points
    Keep the smallest dca
    CUT if one track (or both) doesn't point away from Pvx
    CUT if V0 decays after first hit of either track
    Calulate approximated 3D dca between both helices
    CUT on dcaV0Daughters
    CUT 2 on dcaTrackToPvx if "low" $p_\perp$
    CUT on decay length from Pvx
    CUT if V0 doesn't point away from Pvx
    CUT on dcaV0ToPvx
    CUT on $\alpha_{Arm}$
    CUT on $p_{\perp_{Arm}}$
    Fill an StV0Vertex
    Call UseV0 to find if this V0 is used for Xis
    If primary or used in Xi : store
```

FIG. 4 : *Cuts and algorithm of the V0Finder.*

Before the loops, tables called `ptrks`, `ntrks`, etc..., are filled in the function `StV0FinderMaker::Prepare()`, and the values used afterwards are those that are stored in these tables, in order to improve the speed of the code.

The cuts' values are got in the function `StV0FinderMaker::GetPars()`, and stored in the member objects `pars` and `pars2`. Here are the components :

- `n_point` : number of hits,
- `dcapnmin` : distance of closest approach between the daughter tracks and the primary vertex,
- `dca` : distance of closest approach between the V0 daughters,
- `dlen` : decay length of the V0,
- `dcav0` : distance of closest approach between the V0 trajectory and the primary vertex,
- `alpha_max` : α Armanteros,
- `ptarm_max` : $p_\perp$ Armanteros.

In the description of the algorithm, not that the cut on the number of hits is mentionned twice : the first one is in the code only since March 4th 2004, while the second one has been removed the same day. The effect of the 2nd one is to cut only the V0 daughters, exactly as done in the Fortran code. Yet, cutting also the Xis' bachelors dramatically reduces the background (it's cut by 19 %), so the cut on the number of hits has been moved to where it is now (cut 1), so as to cut the tracks directly before even filling the table of tracks that will be used afterwards in the code. It therefore saves time and memory.

A second cut that wasn't in the Fortran code has been added in the V0Finder : it requires that the V0 decays before the first hit of each of its daughters' tracks.

A third change between Fortran and C++ is the few lines of code that calculate the 3D-dca between both helices. The Fortran-equivalent code has been kept as a comment in the current C++ code, and both blocks (the new one and the Fortran-equivalent one) are clearly mentionned in the V0Finder code.

Some information about each cut applied :

- Number of hits : both tracks must have a number of hits $\geq$ `DB`[1]`->n_point`
- DcaTrackToPvx : dca between each of the tracks and the primary vertex : see explanations in the next paragraph
- Track points away from Pvx : both tracks have to point away from the primary vertex, i.e. if we call X the primary vertex and M the point of a track that is the closest to the other helix, $\overrightarrow{p_M} \cdot \overrightarrow{XM}$ must be positive
- V0 decays after the first hit of either track : this cut removes obviously bad candidates (or bad decay lengths calculations) which have a decay length that is e.g. longer than the size of the TPC. The first hit is assumed to be at `StPhysicalHelix::origin`. Calling it H, and V the V0 decay point, the requirement is that $\overrightarrow{p_{V0_{at\ V}}} \cdot \overrightarrow{VH}$ must be positive
- DcaV0Daughters : the calculated dca between both tracks has to be $<$ `DB->dca`
- Decay length : the calculated V0 decay length has to be $>$ `DB->dlen`. It's actually the distance between the primary vertex and the V0 decay point, so it matches with the decay length only for the primary V0's
- V0 points away from Pvx : calling X the primary vertex and A the point where both helices are closest to each other, $\overrightarrow{p_A} \cdot \overrightarrow{XA}$ must be positive
- DcaV0ToPvx : the calculated dca between the V0 and the primary vertex must be $<$ `DB->dcav0`
- Alpha Armanteros : α_{Arm} must be $\leq$ `DB->alpha_max`

[1]Database.

- Pt Armanteros : $p_{\perp_{Arm}}$ must be $\leq$ `DB->ptarm_max`

Now, the dcaTrackToPvx cut requires some non-obvious explanations. Its second occurence is simple to understand : it is a cut on the dca of both tracks to the primary vertex – both have to be > `DB->dcapnmin` – that is applied only when $p^2_{\perp_{V0}}$ is lower than a variable called `ptV0sq`, and whose value is set to $(3.5\ GeV)^2$ in the constructor of `StV0FinderMaker`. This is done to cut less signal at high-$p_\perp$, an area where there is very few background, thus enabeling a loosening of the cuts.

Its first occurence is done to apply this cut as soon as possible, for the code to be faster (less track pairs to process), at a time when $\overrightarrow{p_{V0}}$ is not calculated yet. The reason why this is possible is that, indexing with n (resp. p) what is related with the negative (resp. postive) daughter,

$$p_{\perp n} + p_{\perp p} \geq p_{\perp_{V0}} \tag{1}$$

Here is the demonstration : let's call r the axis that is parallel to $\overrightarrow{p_{V0}}$, θ the perpandicular axis. With these $(\overrightarrow{u_r}, \overrightarrow{u_\theta})$ coordinates, we have :

$$\begin{cases} p_{V0_r} &= p_{n_r} + p_{p_r} \\ p_{V0_\theta} &= p_{n_\theta} + p_{p_\theta} = 0 \end{cases}$$

i.e. :

$$p_{\perp_{V0}} = p_{V0_r} = p_{n_r} + p_{p_r}$$

Since $p_{\perp n} \geq p_{n_r}$ and $p_{\perp p} \geq p_{p_r}$[1], we obtain :

$$p_{\perp n} + p_{\perp p} \geq p_{n_r} + p_{p_r} = p_{\perp_{V0}} \quad , \qquad \text{QED}^2$$

So the first occurence of the dcaTrackToPvx cut applies this cut when $(p_{\perp n} + p_{\perp p})^2 \leq$ `ptV0sq`, with `ptV0sq` $= 3.5\ GeV$, because according to (1), any track pair cut by this condition would anyway have been cut by the second occurence of the dcaTrackToPvx cut (because $p_{\perp_{V0}} \leq p_{\perp n} + p_{\perp p} \leq 3.5\ GeV$).

For security, a factor of 0.98 multiplies the $p_\perp$ limit in the first occurence of the cut. To sum up :

- First occurence : if $p_{\perp n} + p_{\perp p} \leq \sqrt{0.98} \times 3.5\ GeV$, apply the dcaTrackToPvx cut on both tracks
- Second occurence : if $p_{\perp_{V0}} \leq 3.5\ GeV$, apply the dcaTrackToPvx cut on both tracks

4.4 XiFinder

4.4.1 Differences between Fortran and C++

The differences between the Fortran code and the C++ code are shown in Fig. 5, on the half-detailed algorithm. The red lines show what has disappeared, either because of the new structure of the code, or because of the fact that we are using StEvent. The green lines show the parts that have been reshaped, for the same reasons as several parts have been removed.

Apart from these modifications, the code is a simple translation from Fortran to C++. This may change once we are convinced that the C++ code has no bug : we may then want to have the code more readable, or better organised, or we may even want to replace some calculation algorithms.

So far, the code is all in one block, for speed purposes, and the beginning and end of each former Fortran subroutine is indicated by commented lines. Once again, when we are sure that we don't need to compare the C++ and Fortran codes anymore, we'll probably remove all this.

[1]Because of the opening angle of the V0 decay, the inequality is most often strict.

[2]Quod erat demonstrandum, not quantum electrodynamics ;-) .

Avoiding calls to subfunctions resulted in a duplication of a certain part of the code. Figure 7 shows how the C++ code structure fits to the Fortran one, but let's detail the changes (green lines in FIG. 5) one by one.

```
Do things...
Loop over V0 vertices
  Find vertex key
  Hits in which detectors
  Lambda mass checks, cuts
  Loop over global tracks
    Select correct charge
    Don't use V0 tracks
    Hits in which detectors
    Parameters conversion
    Subroutine circle_param
    Calculate dca V0/bachelor in 2D
    Subroutine casc_geom
    Loop over the 2 intersect. points
      Subroutine update_track_param
      Approxim. of 3D-dca by lineari-
                    sation of the helix
      Subroutine track_mom
      Check validity of linear approx.
      Subroutine ev0_project_track
      If not good enough : try again
                        (up to 3 tries)
      If cuts OK : fill table
```

```
Do things...
Loop over V0 vertices

  Hits in which detectors
  Lambda mass checks, cuts
  Loop over global tracks
    Select correct charge
    Don't use V0 tracks
    Hits in which detectors


    Calculate dca V0/bachelor in 2D
    Subroutine casc_geom
    Loop over the 2 intersect. points
      Subroutine update_track_param
      Approxim. of 3D-dca by lineari-
                    sation of the helix

      Check validity of linear approx.
      Subroutine ev0_project_track
      If not good enough : try again
                        (up to 3 tries)
      If cuts OK : fill StEvent
```

FIG. 5 : *Differences Fortran vs C++ in the XiFinder algorithm.*

The reshaping of the Lambda mass calculation consisted simply in calculating the invariant mass using other parameters : taking the example of the Λ (as opposed to the $\overline{\Lambda}$) invariant mass, we have :

$$\begin{aligned} m_\Lambda^2 &= E_\Lambda^2 - \overrightarrow{p}_\Lambda^2 \\ &= (E_+ + E_-)^2 - (\overrightarrow{p}_+ + \overrightarrow{p}_-)^2 \\ &= E_+^2 + E_-^2 + 2E_+E_- - \overrightarrow{p}_+^2 - \overrightarrow{p}_-^2 - 2\overrightarrow{p}_+\overrightarrow{p}_- \end{aligned}$$

In Fortran, the E and $\overrightarrow{p}$ terms are grouped together, and the invariant mass is calculated with this formula :

$$m_\Lambda^2 = m_p^2 + m_\pi^2 + 2E_+E_- - 2\overrightarrow{p}_+\overrightarrow{p}_-$$

In C++, energies and momenta are kept separated, and the formula used is, as already written above :

$$\begin{aligned} m_\Lambda^2 &= (E_+ + E_-)^2 - (\overrightarrow{p}_+ + \overrightarrow{p}_-)^2 \\ &= \left(\sqrt{\overrightarrow{p}_+^2 + m_p^2} + \sqrt{\overrightarrow{p}_-^2 + m_\pi^2}\right)^2 - \overrightarrow{p}_\Lambda^2 \end{aligned}$$

To modify the structure of *casc_geom* and of the loop coloured in green in FIG. 5, I've written the Fortran code as a set of for-loops, if-loops, goto's and blocks of instructions. The figures below show those reashapings : FIG. 6 shows the reshaping of *casc_geom*, and FIG. 7 shows the reshaping of the inside of the loop over the intersection points (between the bachelor's helix and the V0's straight line). In FIG. 7, what is called *Block 5* is actually a large part of the Fortran program, and it hasn't been reshaped.

```
if ($p_x \neq 0$)
  Block 1
  if ($c < 0$) goto 137
  Block 2
else  //($p_x == 0$)
  Block 3
  if ($c < 0$) goto 137
  Block 4
exit  //(from the subroutine)
Lbl 137
if ($p_x \neq 0$)
  Block 5
else  //($p_x == 0$)
  Block 6
```

$\Longrightarrow$

```
if ($p_x \neq 0$)
  Block 1
  if ($c < 0$) Block 5
    else Block 2
else  //($p_x == 0$)
  Block 3
  if ($c < 0$) Block 6
    else Block 4
```

FIG. 6 : *Reshaping of* `casc_geom` *: Fortran to C++.*

```
tries=1
Block 1
Lbl 60
Block 2
if (cond1 && tries≤3)
  Block 3
  if (cond2)  //(Depends on Block 2)
    tries++
    Block 4
    Goto 60
  Block 5
if (cond3)
  Goto 30  //(Just after the for-loop)
```

$\Longrightarrow$

```
tries=1
Block 1
Block 2
while (cond1 && tries≤3 && cond2)
  tries++
  Block 3
  Block 4
  Block 2
if (cond1 && tries≤3)
  Block 3
  Block 5
if (cond3) break
```

FIG. 7 : *Reshaping of the loop over the intersection points : Fortran to C++.*

4.4.2 Calculation of the intersection points

Now, here is how are calculated the coordinates of the intersection points *in the bending plane (xOy)* between the V0's straight line and the bachelor's helix (actually, of their projection in the bending plane). This is done in the code in the former subroutine `casc_geom`.

Let's call Δ the projection of the V0's trajectory, and $\mathcal{C}$ the circle that is the projection of the bachelor's trajectory. Their equations are :

$$\Delta : y = ax + b \qquad\qquad \mathcal{C} : (x - x_c)^2 + (y - y_c)^2 = R^2$$

Calling (x_0, y_0) a point on the V0's trajectory, and $\overrightarrow{p} = (p_x, p_y, p_z)$ its momentum, we obtain :

$$a = \frac{p_y}{p_x} \qquad\qquad b = y_0 - \frac{p_y}{p_x} x_0$$

Thus :

$$\Delta : y = \frac{p_y}{p_x}(x - x_0) + y_0$$

If we change the variables, with $X = x - x_c$ and $Y = y - y_c$, we have :

$$\Delta : Y = \frac{p_y}{p_x}(X + x_c - x_0) + y_0 - y_c \qquad\qquad \mathcal{C} : X^2 + Y^2 = R^2$$

Now, we can search the intersection points. If we call $\delta_x = x_c - x_0$ and $\delta_y = y_c - y_0$, we have to solve this system of equations :

$$\left\{ \begin{array}{lcl} Y^2 & = & R^2 - X^2 \\ Y^2 & = & \left(\frac{p_y}{p_x}(X + \delta_x) - \delta_y\right)^2 \end{array} \right.$$

which, if we define $\alpha = p_y/p_x$ and $\beta = \alpha\delta_x - \delta_y$, and modify the equations, becomes :

$$\left\{ \begin{array}{l} Y = \alpha(X + \delta_x) - \delta_y \\ X^2(\alpha^2 + 1) + 2\alpha\beta X + \beta^2 - R^2 = 0 \end{array} \right.$$

If the condition $R^2(\alpha^2 + 1) \geq \beta^2$ is true, then we have 2 solutions, that are :

$$\left\{ \begin{array}{l} Y = \alpha(X + \delta_x) - \delta_y \\ X = \frac{\alpha\beta \pm \sqrt{R^2(\alpha^2+1) - \beta^2}}{\alpha^2+1} \end{array} \right.$$

The 2-D coordinates of these 2 points are stored in the code in variables called `xOut` and `yOut`.

4.4.3 Calculation of the dca between the V0 and the bachelor

The algorithm that calculates the dca between the V0 (a line) and the bachelor (a helix) is rather intuitive (see also Fig. 11). This dca can't be calculated analytically, so the trick is to linearise the helix locally. This means that we will assume, for the dca calculation, that the helix is equal to its tangent at the intersection point between the helix and the V0 line.

Then, the position of the point of the tangent where the distance to the V0 line is the smallest is calculated. To check that the linearisation is not a too strong approximation, the distance between that point and the actual helix is calculated, and is required to be smaller than a certain fraction of the helix' radius. If this is true but if the distance is yet bigger than another fraction of the helix' radius (obviously smaller than the previous one), then the helix is linearised at the calculated point (instead of the intersection point in 2-D), and the calculation is re-done.

This is done 3 times, or less if the distance between the calculated point and the actual helix matches the second criterium after less than 3 loops. So all the candidates that match the first criterium are kept, and those not matching the second criterium are simply improved by trying 3 times to linearise the helix at a point that is closer to the actual point where the distance to the V0 line is the smallest.

The first part of the algorithm is illustrated by figure 8, which shows the projection in the plane (xOy) of the cascade geometry : that gives the circle $\mathcal{C}$ (projection of the helix) and the line Δ (projection of the 3-D line). M is one of the 2 intersection points (always in 2-D ; in 3-D, the helix and the line almost never intersect), C is the center of the circle, R its radius, and A is the projection of what is called the origin of the helix (it's most often the first hit point, or more exactly the point of the helix that is closest to the first hit, since the helix is a fit). Ψ is the angle between the x axis and the tangent to the circle in A. For lack of imagination, I'll keep the same names for the non-projected objects later ;-) (i.e. $\mathcal{C}$ for the helix, Δ for the V0 line and A for the origin (not its projection) of the helix).

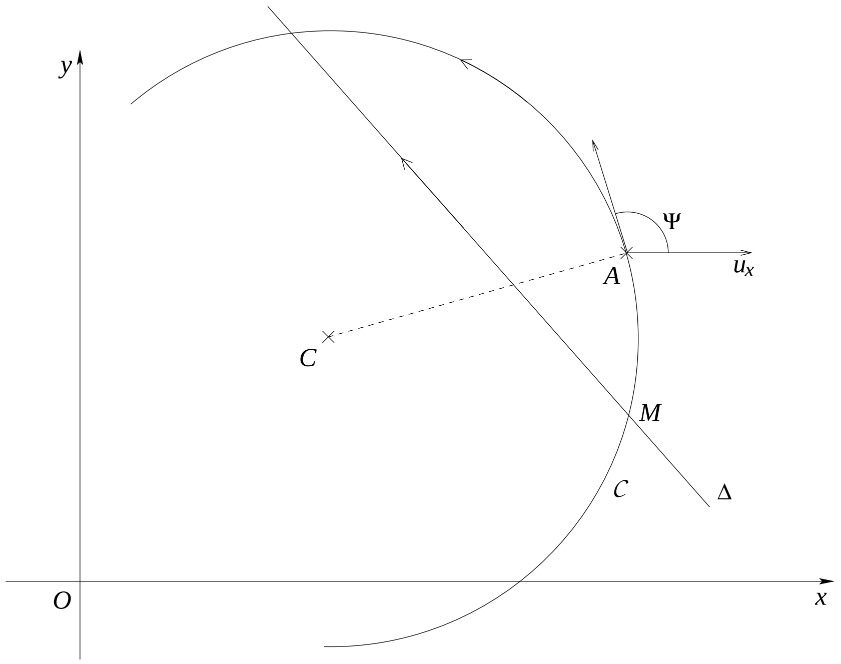


FIG. 8 : *Projection of the cascade geometry in the 2-D plane (xOy).*

The first part of the code that is run is the former subroutine `update_track_param`. Its role is simply to move the orgin of the helix from its former position A to its new position M (actually, the point of the helix that overlaps with M when projected to the (xOy) plane).

Here is a list of the various variables in this area of the code :

- `axb` : $\overrightarrow{CA} \cdot \overrightarrow{w}$, where $\overrightarrow{w}$ is the vector such as $\|\overrightarrow{w}\| = \|\overrightarrow{CM}\|$ and $(\widehat{\overrightarrow{CM}, \overrightarrow{w}}) = -\frac{\pi}{2}$
- `arg` : $\sin(\overrightarrow{CA}, \overrightarrow{CM})$
- `ds` : curvilinear length on the *circle* between A and M
- `dz` : $z_M - z_A$

And what follows is how to do the link between the code and the mathematical formulas :
In the code, calling xi and yi the coordinates of the origin :
`axb = (xi-xc)(yOut-yc) - (yi-yc)(xOut-xc)`

$$\overrightarrow{CA} = \left| \begin{array}{l} x_A - x_C \\ y_A - y_C \end{array} \right. \qquad \overrightarrow{CM} = \left| \begin{array}{l} x_M - x_C \\ y_M - y_C \end{array} \right. \qquad \overrightarrow{w} = \left| \begin{array}{l} y_M - y_C \\ -(x_M - x_C) \end{array} \right.$$

So $\overrightarrow{CA} \cdot \overrightarrow{w} = (x_A - x_C)(y_M - y_C) - (y_A - y_C)(x_M - x_C) =$ `axb`.
Now, let's try to find what is this `axb` $= \overrightarrow{CA} \cdot \overrightarrow{w}$:

we know that $\|\overrightarrow{w}\| = \|\overrightarrow{CM}\|$, so :

$$\begin{aligned}\overrightarrow{CA} \cdot \overrightarrow{w} &= \|\overrightarrow{CA}\| \cdot \|\overrightarrow{w}\| \cos(\overrightarrow{CA}, \overrightarrow{w}) \\ &= \|\overrightarrow{CA}\| \cdot \|\overrightarrow{CM}\| \cos((\overrightarrow{CA}, \overrightarrow{CM}) + (\overrightarrow{CM}, \overrightarrow{w})) \\ &= R^2 \cos((\overrightarrow{CA}, \overrightarrow{CM}) - \tfrac{\pi}{2}) \\ &= R^2 \sin(\overrightarrow{CA}, \overrightarrow{CM})\end{aligned}$$

Therefore, since `rsq` $= R^2$, we obtain `arg` = `axb/rsq` $= \sin(\overrightarrow{CA}, \overrightarrow{CM})$.
`ds` is then defined as the angle (arcsin(`arg`)) multiplied by R, i.e. it is the curvilinear length on the circle from A to M.
And then, from the definition of the dip angle[1], we obtain that `dz = ds.tan(dipAngle)` is $z_M - z_A$ (considering this time A and M as the points on the helix instead of on the circle).

At the end of the former subroutine `update_track_param`, a helix called `bachGeom2` is booked with the same parameters than the original bachelor helix taken from the track container, `bachGeom`, except for the origin that has been moved from A to M, and the angle Ψ that obviously changes when the origin moves (see FIG. 8).

The next piece of code is the former subroutine `track_mom`, which just books the momentum of the bachelor taken in M in the variable `xOrig` (which contained the 3-D position of M a couple of code-lines before : since both usages don't overlap, the same StThreeVector can be used for both of them).

The next part of the code is the most difficult one to understand. It wasn't a subroutine in the Fortran code : that was part of the `exiam` function. Here is a list of the various variables in this area of the code :

- `pBach` : normalised momentum of the bachelor in M (so it's rather the (normalised) direction of the tangent to the helix in M)
- `dv0dotdb` : $\cos(\overrightarrow{\texttt{dpV0}}, \overrightarrow{\texttt{pBach}})$
- `diffc` : $\overrightarrow{MV}$, calling V the point where the V0 decays
- `denom` : $\cos^2(\overrightarrow{\texttt{dpV0}}, \overrightarrow{\texttt{pBach}}) - 1$
- `s2` : ehm... well... see the explanations below !
- `valid` : relative error due to the linearisation

So let's call V the position of the V0 vertex, i.e. the point where the V0 decays. As described p. 14, we now linearise the helix, i.e. we assume that the helix can be merged with its tangent in M (in 3-D). So an approximation of the point where the distance between the helix and the V0 line is the smallest is the point where the distance between the tangent to the helix and the V0 line is the smallest.

Let $\mathcal{D}$ be the tangent to the helix in M, Δ being the V0 line, and let H_1 (resp. H_2) be the point on Δ (resp. on $\mathcal{D}$) where the distance to $\mathcal{D}$ (resp. to Δ) is the smallest. A 3-D illustration with those points can be seen on FIG. 9.

Calling $\overrightarrow{u}$ the vector that drives Δ (i.e. $\overrightarrow{u} = \overrightarrow{\texttt{dpV0}}$) and $\overrightarrow{v}$ the vector that drives $\mathcal{D}$ (i.e. $\overrightarrow{v} = \overrightarrow{\texttt{pBach}}$), we can write H_1 and H_2 as :

$$H_1 = V + k_1 \overrightarrow{u} \qquad\qquad H_2 = M + k_2 \overrightarrow{v}$$

With the definition of H_1 and H_2 above, we can write that we search :

$$(H_1 \in \Delta, H_2 \in \mathcal{D}) \quad / \quad \overrightarrow{H_1H_2} \perp \Delta \quad \text{and} \quad \overrightarrow{H_1H_2} \perp \mathcal{D} \tag{2}$$

[1] For a detailed note about the helices' parameters, see the appendix A of the Star Class Library documentation.

$$\Leftrightarrow \quad (H_1 \in \Delta, H_2 \in \mathcal{D}) \quad / \quad \overrightarrow{H_1H_2} \cdot \overrightarrow{u} = \overrightarrow{H_1H_2} \cdot \overrightarrow{v} = 0 \tag{3}$$

Given that

$$\begin{aligned}\overrightarrow{H_1H_2} &= M + k_2\overrightarrow{v} - V - k_1\overrightarrow{u} \\ &= \overrightarrow{VM} + k_2\overrightarrow{v} - k_1\overrightarrow{u} \\ &= -\overrightarrow{\texttt{diffc}} + k_2\overrightarrow{v} - k_1\overrightarrow{u}\end{aligned}$$

we can re-write the system (3) as

$$\begin{cases} \overrightarrow{H_1H_2} \cdot \overrightarrow{u} = -\overrightarrow{\texttt{diffc}} \cdot \overrightarrow{u} + k_2\overrightarrow{v} \cdot \overrightarrow{u} - k_1\overrightarrow{u} \cdot \overrightarrow{u} = 0 \\ \overrightarrow{H_1H_2} \cdot \overrightarrow{v} = -\overrightarrow{\texttt{diffc}} \cdot \overrightarrow{v} + k_2\overrightarrow{v} \cdot \overrightarrow{v} - k_1\overrightarrow{u} \cdot \overrightarrow{v} = 0 \end{cases}$$

which can also be written as

$$\begin{cases} -\overrightarrow{\texttt{diffc}} \cdot \overrightarrow{u} + k_2 \cos(\overrightarrow{v}, \overrightarrow{u}) - k_1 = 0 \\ -\overrightarrow{\texttt{diffc}} \cdot \overrightarrow{v} + k_2 - k_1 \cos(\overrightarrow{u}, \overrightarrow{v}) = 0 \end{cases}$$

because $\|\overrightarrow{u}\| = \|\overrightarrow{v}\| = 1$.

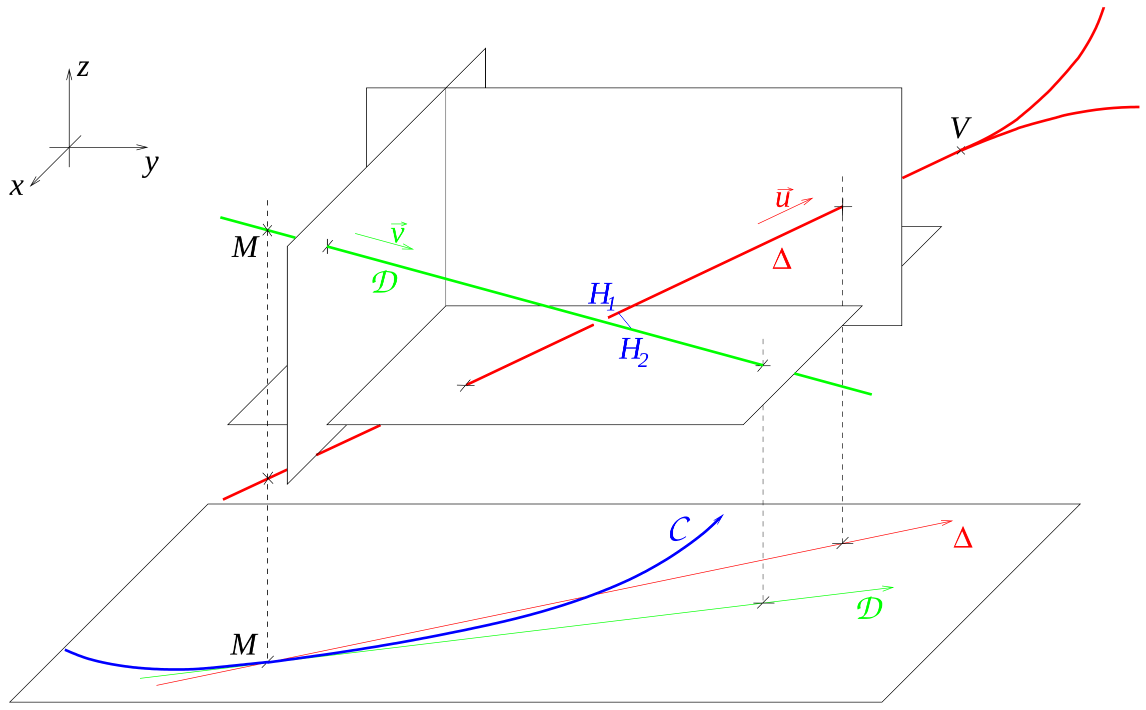


FIG. 9 : *3-D illustration of lines and points' names introduced in the text. The V0 is in red, the projection of the helix in the bending plane is in blue, and the tangent to this helix is in green.*

Solving this system, we obtain :

$$\begin{cases} k_1 = \dfrac{-\overrightarrow{\texttt{diffc}} \cdot (\overrightarrow{v} \cos(\overrightarrow{u}, \overrightarrow{v}) - \overrightarrow{u})}{\cos^2(\overrightarrow{u}, \overrightarrow{v}) - 1} \\ k_2 = \dfrac{\overrightarrow{\texttt{diffc}} \cdot (\overrightarrow{u} \cos(\overrightarrow{u}, \overrightarrow{v}) - \overrightarrow{v})}{\cos^2(\overrightarrow{u}, \overrightarrow{v}) - 1} \end{cases}$$

In the code, `s2` is calculated as :

```
s2 = (dpV0.X dv0dotdb - pBach.X) diffc.X + (dpV0.Y dv0dotdb - pBach.Y) diffc.Y +
     + (dpV0.Z dv0dotdb - pBach.Z) diffc.Z;
s2 = s2/denom;
```

which can be written as[1] :

$$\begin{aligned}\texttt{s2} &= \frac{(\overrightarrow{\texttt{dpV0}}\cos(\overrightarrow{u},\overrightarrow{v})-\overrightarrow{\texttt{pBach}})\cdot\overrightarrow{\texttt{diffc}}}{\cos^2(\overrightarrow{u},\overrightarrow{v})-1}\\ &= \frac{\overrightarrow{\texttt{diffc}}\cdot(\overrightarrow{u}\cos(\overrightarrow{u},\overrightarrow{v})-\overrightarrow{v})}{\cos^2(\overrightarrow{u},\overrightarrow{v})-1}\\ &= k_2\end{aligned}$$

So `s2` is the 3-D algebraic distance $\overline{MH_2}$ between M and H_2, the point of $\mathcal{D}$ that is closest to Δ.

Then, `valid` is calculated as $\left|\texttt{s2}\sqrt{\texttt{pBach.X}^2+\texttt{pBach.Y}^2}\right|$, i.e. it's the distance in the (xOy) plane between M and H_2, as illustrated by figure 10. The value `valid` itself is not very helpful to determine if the linearisation is a good approximation or not. The value that has to be looked at is actually the distance between H_2 and the circle in the 2-D plane, which is called d in figure 10.

But d actually depends explicitely on `valid`, which means that an initial requirement on d can be transformed into a requirement on `valid`. This allows not to calculate d and saves some calculation time – at least it's the only reason I've found that would explain why the authors of the code have chosen to test `valid` rather than d ! The relation between d and `valid` is :

$$d = \sqrt{R^2+\texttt{valid}^2} - R$$

$$\Leftrightarrow \quad \frac{d}{R} = \sqrt{1+\left(\frac{\texttt{valid}}{R}\right)^2} - 1 \tag{4}$$

There are 2 conditions on $\frac{\texttt{valid}}{R}$. Let's call these two values $valid_1$ and $valid_2$, with $valid_1 < valid_2$. The original algorithm (it may be changed in the mid-future) throws away any Xi candidate for which `valid` $> valid_2$, and keeps all the other ones. But if `valid` $\in [valid_1; valid_2]$, then another part of the algorithm, which is described in the next paragraph, is run. It consists in improving the quality of the linearisation by linearising the helix at another point. This improvement is tried at most 3 times. Basically, this means that a linearisation that gives `valid` $> valid_2$ means that it's hopelessly bad ; when `valid` $< valid_1$ it means that the linearisation is good enough and doesn't need to be improved ; and when `valid` $\in [valid_1; valid_2]$, the linearisation is improved but it actually doesn't matter if the criterium `valid` $< valid_1$ is not reached : the candidate is kept anyway. According to the tests, only a very few proportion of the candidates need 3 passes in the loop[2], so requiring more than 3 passes is indeed not necessary.

This table shows the numerical values of $valid_1$ and $valid_2$ used in the code (first line) and their equivalent for the more interesting variable $\frac{d}{R}$, calculated with (4).

$0.001\ R$	$<$	`valid`	$<$	$0.02\ R$
$5.10^{-7}\ R$	$<$	d	$<$	$2.10^{-4}\ R$
$0.0005\ \%$	$<$	$\frac{d}{R}$	$<$	$0.02\ \%$

So when `valid` $\in [valid_1; valid_2]$, here is the piece of code that is run :

[1]Given that `dv0dotdb` $= \overrightarrow{\texttt{dpV0}}\cdot\overrightarrow{\texttt{pBach}} = \cos(\overrightarrow{\texttt{dpV0}},\overrightarrow{\texttt{pBach}})$. Thus `denom` $= \cos^2(\overrightarrow{\texttt{dpV0}},\overrightarrow{\texttt{pBach}}) - 1 = \cos^2(\overrightarrow{u},\overrightarrow{v}) - 1$.

[2]Result obtained over 1 Au-Au 200 *GeV* central event : over 73 269 bachelors, neglecting those which have only 1 intersection point, 45 500 have 2 intersection points, i.e. 62 % of them (and therefore 38 % have no intersection points). Among the 118 194 dca calculations of those 62 %, 77.0 % of them don't need a better linearisation, 21.7 % need to go once in the loop, 0.8 % need to go twice in the loop, and 0.5 % go 3 times in the loop (this latter percentage, unike the former ones, is the number of candidates that need only 3 passes added to the number of candidates that would need more).

- Former subroutine `ev0_project_track` : calculates the coordinates (in the (xOy) plane) of the point M' defined as the intersection of the circle $\mathcal{C}$ and the line (CH_2) (see FIG. 10)
- Former subroutine `update_track_param` : moves the origin of the helix from M to M' (as previously done from A to M)
- Former subroutine `track_mom` : calculates the momentum of the bachelor in M' (as previously done in M)
- Block that calculates `s2` and `valid` : calculates a new `s2` and `valid`, whose value will be checked to see if one more pass in this loop is necessary

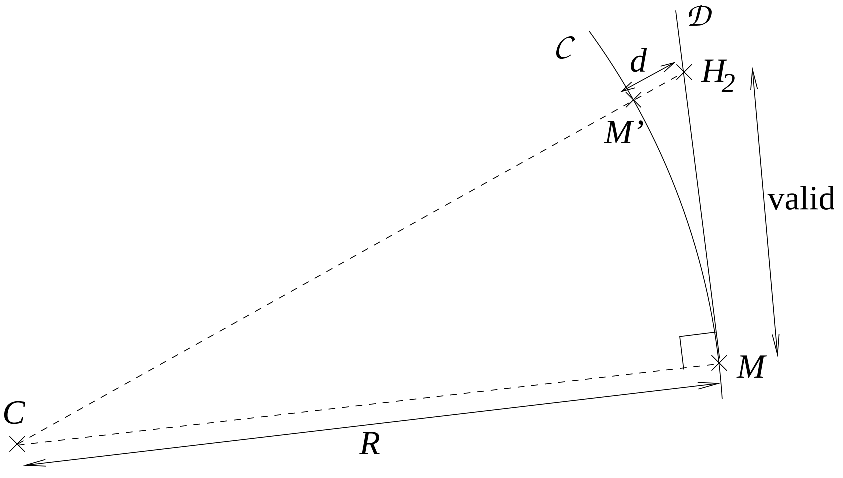


FIG. 10 : *2-D plane evaluation of the quality of the approximation made by linearising the helix.*

The 3 last blocks are exactly the same as those already described above, so I'll only describe the former subroutine `ev0_project_track` : the list below is made of the variables that are used in this area of the code :

- `batv` : 3-D coordinates of H_2
- `dtmp` : $x_C - x_{H_2}$
- `atmp` : $y_C - y_{H_2}$
- `ctmp` : slope of the line (CH_2)
- `yy` : $y_{M'} - y_C$
- `zz` : $x_{M'} - x_C$
- `xAns` : temporary variable, actually equal to xOut
- `yAns` : temporary variable, actually equal to yOut
- `xOut` : $x_{M'}$ (contained x_M before)
- `yOut` : $y_{M'}$ (contained y_M before)

The calculation of $(x_{M'}, y_{M'})$ is simple : since $\texttt{ctmp} = \frac{x_C - x_{H_2}}{y_C - y_{H_2}}$ is the inverse of the slope of (CH_2) the equation of (CH_2) is :

$$(CH_2) : y = \frac{1}{\texttt{ctmp}}(x - x_C) + y_C$$

and therefore, setting $x' = x_{M'} - x_C$ and $y' = y_{M'} - y_C$, M' is such as :

$$\left\{ \begin{array}{l} y' = \left(\frac{1}{\texttt{ctmp}}\right) x' \\ {x'}^2 + {y'}^2 = R^2 \end{array} \right.$$

Solving this system gives :

$$\begin{cases} y' = \pm\frac{R}{\sqrt{1+\texttt{ctmp}^2}} \\ x' = \texttt{ctmp}\cdot y' \end{cases}$$

x' and y' are respectively `zz` and `yy` in the code, and the "signus dilemma" is solved by the if-loop on the sign of `atmp`.

And here is eventually the list of the variables used at the end of the code :

- `v0atv` : 3-D coordinates of H_1
- `s1` : see the explanations below and page 16
- `xOrig` : momentum of the bachelor at H_2 (be careful : at other places of the code, `xOrig` actually stores the origin of the bachelor's helix)
- `dca` : $d^2_{H_1H_2} = \left\|\overrightarrow{\texttt{v0atv} - \texttt{batv}}\right\|^2$: it's the (squared) approximated distance of closest approach between the Lambda and the bachelor. Keeping the squared value and taking the root only at the moment of storing the value avoids time-consuming square-root calculations before all the final cuts
- `xpp` : position of the Xi decay point : it's the middle of $[H_1H_2]$, i.e. $\frac{\texttt{v0atv}-\texttt{batv}}{2}$
- `rv` : the Xi decay length ($\left\|\overrightarrow{\texttt{xpp} - \texttt{xPvx}}\right\|$), used only in the former versions of the code (and in Fortran). Now, the cut on the decay length is done "on the fly"
- `pXi` : momentum of the reconstructed Xi : $\overrightarrow{p_{Xi}} = \overrightarrow{p_{V0}} + \overrightarrow{p_{Bach,H_2}}$
- `check` : used only in Fortran and in the former versions of the C++ code : contains first $\overrightarrow{H_1V}\cdot\overrightarrow{p_{V0}}$ ((`xV0` - `v0atv`) . $\overrightarrow{\texttt{pV0}}$), and then $\overrightarrow{PvxXivtx}\cdot\overrightarrow{p_{Xi}}$ ((`xpp` - `xPvx`) . $\overrightarrow{\texttt{pXi}}$) to check that the V0 points away from the Xi vertex and the Xi points away from the primary vertex
- `pper` : $p_\perp$ Armanteros of the Xi
- `globHelix` : temporary `StHelix` booked with temporary variables (their name end in `_tmp`), used to calculate `bxi` with StEvent functions
- `bxi` : distance of closest approach between the Xi and the primary vertex

At the end of the while-loop, the candidates for which `valid` $> valid_2$ are simply thrown away, and all the other ones are kept, even if `valid` $> valid_1$. The value of `s2` calculated during the last loop is kept, and `s1` (which is the k_1 of equation (2) (p. 16) and of the following ones) is calculated as

$$\texttt{s1} = \frac{-\overrightarrow{\texttt{diffc}}\cdot(\overrightarrow{\texttt{pBach}}\times\texttt{dv0dotdb} - \overrightarrow{\texttt{dpV0}})}{\texttt{dv0dotdb}^2 - 1} = \frac{-\overrightarrow{\texttt{diffc}}\cdot(\overrightarrow{v}\cos(\overrightarrow{u},\overrightarrow{v}) - \overrightarrow{u})}{\cos^2(\overrightarrow{u},\overrightarrow{v}) - 1} = k_1$$

Then, the 3-D coordinates of H_1 are calculated and stored in `v0atv`, just like the coordinates of H_2 are stored in `batv`. Once this is done, we check that $\overrightarrow{H_1V}$ goes roughly in the same direction as $\overrightarrow{p_{V0}}$, i.e. that the V0 points away from the Xi vertex newly found. This was formerly done via the variable `check`, that is exactly $\overrightarrow{H_1V}\cdot\overrightarrow{p_{V0}}$, but is now done "on the fly".

If `check` is positive, it means that the V0 points to the opposite direction than where the Xi vertex is, and the rest of the algorithm – run only in that case – can be roughly summed up as something that looks like this for the latest versions of the code :

```
Calculate some cut variables
if (cuts are not OK) continue
pXi = pV0 + xOrig
if (Xi doesn't point away from primary vertex) continue
Calculate $p_\perp$ Armanteros, cut on it
Calculate signed dcaXiToPrimaryVertex, cut on it
Store the candidate
```

and like that for the former versions of the code as well as for the Fortran program :

```
dca = $\left\|\overrightarrow{\texttt{v0atv} - \texttt{batv}}\right\|$            // dca = $H_1H_2$
xpp = $\frac{\texttt{v0atv}-\texttt{batv}}{2}$                 // xpp = 3-D coordinates of the middle of $[H_1H_2]$
rv = $\left\|\overrightarrow{\texttt{xpp} - \texttt{xPvx}}\right\|$              // rv = Xi decay length
if (cuts are OK)
  pXi = pV0 + xOrig                 // $\overrightarrow{p_{Xi}} = \overrightarrow{p_{V0}} + \overrightarrow{p_{Bach,H_2}}$
  check = (xpp - xPvx) . $\overrightarrow{\texttt{pXi}}$         // Check that $\overrightarrow{PvxXivtx} \cdot \overrightarrow{p_{Xi}} \geq 0$, i.e. that the
  if (check $\geq$ 0)                 //       reconstructed Xi points away from the Pvx
    iflag=0 if the dcaXiToPrimaryVertex (cut) is OK
  if (iflag=0)
    Calculates the kinematic variables
    Calculate unsigned dcaXiToPrimaryVertex, cut on it
    Store the candidate
```

4.4.4 Detailed algorithm and cuts

The detailed algorithm is actually explained in the section 4.4.3 concerning the calculation of the dca, p. 14. Yet, in the latter paragraph, no overview of the algorithm is given and the cuts are not listed. This is the purpose of this short paragraph, and is summed up in FIG. 11.

```
Prepare things via the V0Finder
**Loop** over V0 vertices
  Get event- and V0-wise information
  **CUT** on the (false) V0 decay length (distance to primary)
  Prepare rotating variables
  Calculates Lambda invariant mass ; decide if $\Lambda$ or $\overline{\Lambda}$
  **CUT** on $\Lambda$ invariant mass
  Prepare like-sign variable (change charge)
  **Loop** over global tracks
    **If** *track has wrong charge* : next
    Select TPT vs ITTF tracks
    **If** *track already used in the V0* : next
    Determine Xi's detector ID
    Find DCA between V0's straight line and track's helix
            (Implies **CUT** on number of intersection points
                      and **CUT** on validity of linearisation)
    **CUT** if V0 doesn't point away from Xi vertex
    **CUT** on dca between V0 and bachelor
    **CUT** on Xi decay length
    **CUT** if Xi doesn't point away from primary vertex
    **CUT** on $p_\perp$ Armanteros
    **CUT** on dcaXiToPrimaryVertex
    Fill an StXiVertex
    Store it
```

FIG. 11 : *Cuts and algorithm of the XiFinder.*

A short comment : variable `charge` is the sign of the bachelor, so when no like-sign is required, `charge` is also the sign of the Xi : `charge` $= -1$ for Ξ and Ω, $+1$ for $\overline{\Xi}$ and $\overline{\Omega}$.

As for the V0Finder, a cut has been introduced compared to the Fortran version, on Feb 5th 2004 : the cut on the $p_\perp$ Armanteros. It removes 19 % of the candidates (all background), so this cut used with the cut on the number of hits of the bachelor's track[1] remove 34 % of the background.

5 Magnetic field

The magnetic field is calculated from the first track stored in the track table, in `StV0FinderMaker::-Prepare()`. Its sign is simply taken from the charge and the helicity :

$$\mathrm{sgn}(B) = -\mathrm{sgn}(charge) \times \mathrm{sgn}(helicity)$$

while the absolute value is calculated from the momentum of the track : an arbitrary value is given to the magnetic field (1.10^{-10} in the current version of the code) ; then, assuming this value for $\overrightarrow{B}$, the momentum of the track is calculated from its geometrical characteristics (curvature of the helix), by using a function of StEvent. This value is compared to the actual momentum of this track, and the ratio between both momenta simply gives the factor to apply to the arbitrary value given to $\overrightarrow{B}$ to get the real magnetic field :
$B = 1.10^{-10}$
$\overrightarrow{p_1}$ = actual momentum of the particle
$\overrightarrow{p_2}$ = momentum associated to the track assuming B
Actual $B = B \times \frac{p_1}{p_2}$

The magnetic field algebraic value is stored in the member-variable `Bfield`. The value of `tesla` is 1.10^{-13}, that of `kilogauss` is 1.10^{-14}.

If one wants to get the magnetic field value from gufld, it's still possible : the lines mentionned above need of course to be commented out, and 3 pieces of codes just need to be uncommented :

- 3 lines (`include`, `extern`, `define`) at the very beginning of `V0Finder.cxx`
- function `InitRun` in `V0Finder.cxx` and `V0Finder.h`

and Geant has to be instantiated in the macro that calls the V0/XiFinders (if you use the BFC : no worry, it works right away).

6 Detectors

Each track has a *detector ID*, an integer that tells in which detectors this track has been seen. The detector ID of each track is stored in the table `detId`[2] (filled in `StV0FinderMaker::Prepare()`).

The table below sums up the various states of this flag, for tracks that have a hit in the TPC only, in the SVT only, or in both detectors.

[1]Implemented in the V0Finder, although it concerns only the XiFinder. See § 4.3 p. 10 for more information.
[2]Not to be mixed up with table `trkID`, in which the *keys* of the tracks are stored.

`detId`	Hit(s) in TPC	Hit(s) in SVT
1	X	
2		X
3	X	X

Case 2 should not happen this days since the SVT doesn't create tracks up to now, but this may become possible, so the possibility for SVT-only tracks is kept. In this paragraph, "SVT" actually means "SVT and/or SSD".

The detector ID of the V0 and of the Xi are respectively stored in the member-variables `det_id_v0` and `det_id_xi`. Both of them are equal to the higher of their 2 daughters' detector ID. This means that for example a Xi will have a detector ID of 3 whatever the number of its 3 daughters that have at least 1 hit in the SVT (provided that at least one of them has a hit in the SVT).

6.1 Cut values

The cut values depend on various things :

- The collision system : of course, way looser cuts are needed for e.g. `p-p` than for `Au-Au`
- The detector ID : e.g. the position of a Xi vertex is better determined when the SVT is used
- For the V0s : the fact that they are primary or secondary ; the cut on the false decay length (distance between the primary vertex and the V0 decay point) is tighter for primary V0's, because we want them to come from the primary vertex.

The numerical values of the cuts can be found in several files that are stored in directory `StarDb/global/vertices/`. There is one file per collision system and energy. Files for the V0 cuts have a name that begins by *ev0par2*, and those for the Xi cuts have a name that begins by *exipar*.

A V0 file has 2 times 3 sets of cuts. The first 3 sets are cuts used for all the V0s (primary and secondary), and the last 3 sets are tighter cuts used only for the primary V0s. The 3 sets are for (in this order) :

- Tracks that have hits only in the TPC
- Tracks that have hits only in the SVT
- Tracks that have hits in both the TPC and the SVT

A Xi file has only 3 sets of cuts, because all the Xis are primary. The 3 sets are also TPC-only cuts, SVT-only cuts and TPC+SVT cuts.

Here is the composition of each set of cuts, for the V0's (*dca* stands for *distance of closest approach*) :

- `dca` : maximum value of the dca between the two daughter tracks ;
- `dcav0` : maximum value of the dca between the V0 and the primary vertex (impact parameter) ;
- `dlen` : minimum value of the V0 false decay length (distance from the primary vertex to the decay point) ;
- `alpha_max` : maximum absolute value of α_{Arm} ;
- `ptarm_max` : maximum value of $p_{\perp Arm}$;
- `dcapnmin` : minimum value of the dca between the daughter tracks and the primary vertex (impact parameter) ;
- `iflag` : not used ;
- `n_point` : minimum number of points on the track.

And here is that for the Xi's :

- `use_pid` : not used ;

- `dca_max` : maximum value of the dca between the two daughters ;
- `bxi_max` : maximum value of the dca between the Xi and the primary vertex (impact parameter) ;
- `rv_xi` : minimum value of the Xi decay length ;
- `rv_v0` : minimum value of the V0 false decay length (distance from the primary vertex to the decay point) ;
- `dmass` : cut the V0 invariant mass : keep a window of $\pm$ dmass around the mass of the Lambda ;
- `bpn_v0` : minimum value of the dca between the pion daughter of the V0 and the primary vertex (impact parameter) ;
- `pchisq` : not used.

In the code, the Xi cut values are stored in the member-variable `parsXi`, and the V0 cut values are stored in `pars` and `pars2`, respectively for the primary V0's and for all (primary and secondary) the V0's, both being member-variables.

6.2 Using the SVT hits

To be written.

7 Rotating and like-sign

These options have been implemented for analysis purpose, and have been thought to be a plug-and-play code, avoiding private dirty versions ;-) and waste of time for those who wish to use such methods and would have had to code them themselves.

7.1 Like-sign analysis

For a decay channel $A \longrightarrow B + C$, the like-sign method consists in reconstructing A by associating B and $\overline{C}$ rather than B and C. Its name comes from the usage in decays such as $\Lambda \longrightarrow p^{(+)} + \pi^-$ (using like-sign, p would be combined with π^+, a particle that has the same sign), but let's extend this usage to the Ξ-like decays, e.g. $\Xi^- \longrightarrow \Lambda^0 + \pi^-$. Since the Λ is neutral, associating a Λ with a π^+ instead of a π^- isn't doing like-sign strictly speaking, but I will call this like that.

Like-sign analysis has been implemented only in the XiFinder, not in the V0Finder. The reason why is that one of the Xi decay is neutral, and combining a straight line with a positive helix rather than a negative one doesn't change anything. But things are different for a V0 decay : in such a case, "like-singing" means combining 2 helices of same charge, instead of opposite charge, and, although I've never checked that or looked at distributions made by somebody else, I'd bet that things like the combinatoric, the dca distribution, etc... are changed, which means that the background built with this method would be different than the real background, because the same cuts are applied to distributions that don't have the same shape. So in the case of the V0 decays, the rotating method is probably safer than like-sign.

As said previously, the like-sign method in the XiFinder consists in finding candidates built with one of the daughters being the antiparticle of the expected daughter. So we find and store $\Lambda + \pi^+$ (resp. K^+ for the Ω) and $\overline{\Lambda} + \pi^-$ (resp. K^-).

In the post-reconstruction analysis codes, one should be very careful when using like-signed candidates, because the charge of a particle is determined with the charge of the bachelor[1]. So what your code will assume is an Ω^- is a particle made of a $\pi-$ and a $\overline{\Lambda}$ instead of a Λ. So when applying the cuts, all the functions that make a hypothesis on the particle identification should be changed (e.g. `massLambda()` $\rightarrow$ `massAntiLambda()`).

Deciding whether like-sign should or shouldn't be used is done by using the function `StV0FinderMaker::SetLikesignUsage`. The 2 possible values are :

- `kLikesignUseStandard` $= 0$: the standard reconstruction with no like-sign is performed,
- `kLikesignUseLikesign` $= 2$: the like-sign reconstruction is done.

Section 8 p.27 explains how to use these functions.

In the code, like-sign analysis is done in a very easy way :

- When checking that-and-whether the V0 is a Λ or a $\overline{\Lambda}$, -1 is stored in variable `charge` if it's a Λ (i.e. a Ξ^- or Ω^- candidate will be built), +1 if it's a $\overline{\Lambda}$ (i.e. a $\overline{\Xi}^+$ or $\overline{\Omega}^+$ candidate will be built).
- Then, the variable `charge` is transformed according to this formula :
  `charge=-(useLikesign-1)*charge;` :
  if no like-sign has been asked, the variable is unchanged, whereas if like-sign has been required, the sign of `charge` is changed.
- The XiFinder eventually loops over tracks whose charge is such as `charge` $\times$ `track.charge` > 0.

And that's all !

7.2 Rotating analysis

For a decay channel $A \longrightarrow B + C$, the rotating method consists in reconstructing A by associating B with C' rather than C, where C' is the track of a C-like particle whose parameters have been changed. The various possible changes are :

- Rotating : a track is rotated by 180° around the axis that is parallel to (Oz) and that goes through the primary vertex,
- Symmetry : a track is transformed into its symmetric with respect to the (xPy) plane, P being the primary vertex,
- Rotating + symmetry : doing both transformations together, which is equivalent to taking the symmetric of the track with respect to the primary vertex.

Rotating – which will refer from now to all 3 methods described in the previous paragraph – hasn't been implemented in the V0Finder yet, but will be some day (I haven't received any request yet ;-)).

Deciding whether one of these methods should be used and which one is done by using the function `StV0FinderMaker::SetRotatingUsage`. The usage is explained in section 8 p. 27. There are 4 possible values, which are :

- `kRotatingUseStandard` $= 0$: the standard reconstruction with no rotating is performed,
- `kRotatingUseRotating` $= 1$: the bachelor tracks are rotated,
- `kRotatingUseSymmetry` $= 2$: the bachelor tracks are "symmetrised" with respect to the plane that is parallel to the bending plane and goes through the primary vertex,
- `kRotatingUseRotatingAndSymmetry` $= 3$: the bachelor tracks are "symmetrised" with respect to the primary vertex.

[1]This can be found in `StRoot/StStrangeMuDstMaker/StXiMuDst.cc`, in function `StXiMuDst::FillXi(StXiVertex* xiVertex)`.

Unlike with like-sign, nothing has to be changed in the post-reconstruction analysis codes.

Here is a mathematical description of how the various rotating-like methods are performed (see footnote 1 p. 16 about the helices' parameters) : the table below shows how the various helix parameters are modified depending on the method used.

Original helix		Rotating	Symmetry	Both
Charge	c	c	c	c
Angle	Ψ	$\Psi + \pi$	Ψ	$\Psi + \pi$
Curvature	κ	κ	κ	κ
Dip angle	λ	λ	$-\lambda$	$-\lambda$
X origin	x_0	$2x_{Pvx} - x_0$	x_0	$2x_{Pvx} - x_0$
Y origin	y_0	$2y_{Pvx} - y_0$	y_0	$2y_{Pvx} - y_0$
Z origin	z_0	z_0	$2z_{Pvx} - z_0$	$2z_{Pvx} - z_0$
Helicity	h	h	h	h
X momentum	p_x	$-p_x$	p_x	$-p_x$
Y momentum	p_y	$-p_y$	p_y	$-p_y$
Z momentum	p_z	p_z	$-p_z$	$-p_z$

In the code, all rotating methods calculations are achieved in one shot, thanks to the pre-definition of a couple of interesting variables that are described below. In the code, once the bachelor helix is moved, nothing else is changed by the use of a rotating-like method. So what is done is simply the booking (and then usage) of a StHelixModel called `bachGeom` from both the initial parameters of the helix and the "interesting variables", whose value is set at the beginning of the XiFinder, before the loop on the bachelor tracks.

Here is how the helix parameters are modified before their storage in `bachGeom` :

charge	$\longrightarrow$	charge
helicity	$\longrightarrow$	helicity
curvature	$\longrightarrow$	curvature
psi	$\longrightarrow$	psi + **cstPsi**
dipAngle	$\longrightarrow$	**epsDipAngle** × dipAngle
origin.X	$\longrightarrow$	**cstOrigin.X** + **epsOrigin.X** × origin.X
origin.Y	$\longrightarrow$	**cstOrigin.Y** + **epsOrigin.Y** × origin.Y
origin.Z	$\longrightarrow$	**cstOrigin.Z** + **epsOrigin.Z** × origin.Z
momentum.X	$\longrightarrow$	**epsMomentum.X** × momentum.X
momentum.Y	$\longrightarrow$	**epsMomentum.Y** × momentum.Y
momentum.Z	$\longrightarrow$	**epsMomentum.Z** × momentum.Z

The "interesting variables" are written in bold, and the values they are given are listed in the table below (an empty space means that the value is the same as for "no rotating").

The combination of the values in this table and the transformations listed in the previous paragraph give the mathematical transformations listed in the table p. 26. This avoids a check of the rotating choice by an if-loop inside the for-loop on the tracks, and thus time-consuming jumps in the code.

Variable	No rotating	Rotating	Symmetry	Both
`cstPsi`	0	π		π
`epsDipAngle`	+1		-1	-1
`cstOrigin.X`	0	$2x_{Pvx}$		$2x_{Pvx}$
`cstOrigin.Y`	0	$2y_{Pvx}$		$2y_{Pvx}$
`cstOrigin.Z`	0		$2z_{Pvx}$	$2z_{Pvx}$
`epsOrigin.X`	+1	-1		-1
`epsOrigin.Y`	+1	-1		-1
`epsOrigin.Z`	+1		-1	-1
`epsMomentum.X`	+1	-1		-1
`epsMomentum.Y`	+1	-1		-1
`epsMomentum.Z`	+1		-1	-1

8 How to use the V0/XiFinder

Because the V0/XiFinders read StEvent, they are able to take various input files :

- *daq files* : it's then run in the BFC,
- *event.root files* : it's run stand-alone,
- *MuDst.root files* : it's also run stand-alone, and converts the muDst into an StEvent.

In all the cases, you have to add in your macro :
```
gSystem->Load("StSecondaryVertexMaker");
```

8.1 Running in the BFC

BFC options have been set up for the chain to include the secondary vertices makers ; their names and actions are summed up in the table FIG. 12.

The important thing to note is that you should NOT use options *"V02"* and *"Xi2"* at the same time (or *"V0svt"* and *"Xisvt"*), because 2 identical makers would be instantiated. Just don't forget that if you run with option *"Xi2"*, both the V0's and the Xi's will be found and stored.

Yet, you can use e.g. options *"V0"* and *"V02"* at the same time, so as to get both the Fortran and the C++ V0's.

		Fortran			C++			SVT
Name	Maker run	Kinks	V0s	Xis	Kinks	V0s	Xis	usage
Kink	StKinkMaker	X						
V0	StV0Maker		X					
Xi	StXiMaker			X				
Kink2	StKinkMaker				X			
V02	StV0FinderMaker					X		
Xi2	StXiFinderMaker					X	X	
V0svt	StV0FinderMaker					X		X
Xisvt	StXiFinderMaker					X	X	X

FIG. 12 : *BFC options dealing with strangeness reconstruction.*

Notice that the Fortran makers (i.e. the C++ makers that call the Fortran PAMs) are located in `StRoot/St_dst_Maker/`, while the C++ makers are in `StRoot/StSecondaryVertexMaker/`.

More details can be found here :
`www.star.bnl.gov/STAR/comp/pkg/dev/StRoot/StBFChain/doc/`

If you want to run your personal V0/XiFinder in the BFC, you just have to copy `bfc.C` in your `StRoot/macros/`, and change the part mentionned as *"Insert your maker"* so as to have the StV0/XiFinderMaker be run.

8.2 Running stand-alone

If you want to write a strangeness muDst while reading event.root files, you can copy `makeStrange-MuDst.C` in your `StRoot/macros/`, add the instantiation of a StV0/XiFinderMaker in function `run()`, between the instantiation of St_db_Maker and that of StStrangeMuDstMaker ; check that the time stamp is OK (for the cuts values), and that's all. If you read MuDst.root files, the macro is only very slightly different, an example of it will be put in `StRoot/StSecondaryVertexMaker/` soon. One more necessary thing to do if your input files are MuDst.root files : the *EventModelUsage* option has to be set to 1 (see § 8.3.4).

8.3 Modifying the default parameters

When running in the BFC, I don't know how the default parameters can be changed. I usually have my private StXiFinderMaker version with modified default options in the constructor. If somebody knows, feel free to re-write this paragraph !

When running as a stand-alone maker, it's very easy : you just need to call the functions that are described in the table FIG. 3 8, whose name begins by *Set* of course, with an integer argument. You can find the suitable values of these arguments in the code of `StV0FinderMaker.h`.

8.3.1 LikesignUsage and RotatingUsage

LikesignUsage and RotatingUsage are described in details in § 7.

8.3.2 TrackerUsage

TrackerUsage gives the possibility of choosing to use TPT and/or ITTF tracks for the V0 and cascade reconstruction. It has 3 possible values :

- `kTrackerUseTPT` = 0 : only TPT tracks are used,
- `kTrackerUseITTF` = 1 : only ITTF tracks are used,
- `kTrackerUseBOTH` = 2 : all the tracks are used.

When the value of the tracker usage is set to `kTrackerUseBOTH`, all the tracks are used but they are never mixed together, i.e. a V0 will have either 2 TPT daughters, or 2 ITTF daughters, never one daughter of each type. This allows to define TPT and ITTF V0's and Xi's. They can be differentiated afterwards by the fact that the TPT V0's and Xi's have a positive distance of closest approach between the V0 daughters, while this value is negative for the ITTF V0's and Xi's (so be careful in your analysis code...).

Knowing if a track is a TPT track or an ITTF track is done by quering `StTrack->fittingMe-thod()`, and checking if it's equal to a specific value that signs the ITTF and TPT tracks. These values are stored in the member-variables `ITTFflag` and `TPTflag`, and are defined in `pams/global/inc/`, in files `StTrackMethod.h` and `StTrackDefinitions.h`.

8.3.3 SVTUsage

To be written.

8.3.4 EventModelUsage

This option switches between an StEvent-like input and a MuDst-like input. If the input is a microDst, it's first transformed into an StEvent object, before running the code just as if the input was StEvent-like (case of the daq and event.root files). It has therefore 2 possible values :

- `kUseStEvent` $= 0$: takes an StEvent as input,
- `kUseMuDst` $= 1$: takes a muDst as input.

8.3.5 LanguageUsage

Actually, 3 options belong to this category : *LanguageUsage*, *V0LanguageUsage*, and *XiLanguageUsage*. Only 2 of them are used in the code : *V0LanguageUsage* and *XiLanguageUsage* ; *LanguageUsage* is used for simple configurations (those that occur most often) by overwriting the 2 other options.

Those options have initially been set only for tests purposes (comparison between the Fortran and C++ codes), but may be used for other aims ; it's used for example when the SVT hits are used in V0/Xi finding.

The thing to know is that the Fortran V0's and Xi's will always pre-exist the C++ ones, because they are found when things are still table-like (as opposed to StEvent-like). So the possibilities are : keeping or not the Fortran candidates, and finding or not the C++ candidates. Because of this pre-existence, the Xi's found by the C++ code can be built with either a Fortran V0 (if they are kept) or a C++ V0.

Here are the various values taken by LanguageUsage :

- `kLanguageUseSpecial` $= 0 = \overline{000}^2$: used when special V0LanguageUsage and XiLanguageUsage options have to be set (no overwriting of these options)
- `kLanguageUseOldRun` $= 1 = \overline{001}^2$: runs an "old run", i.e. stores only the Fortran V0's and Xi's
- `kLanguageUseRun` $= 2 = \overline{010}^2$: runs a normal run, i.e. stores only the C++ V0's and Xi's (made of C++ V0's of course)
- `kLanguageUseTestV0Finder` $= 5 = \overline{101}^2$: runs a V0Finder test, i.e. stores the Fortran and C++ V0's (and the Fortran Xi's)
- `kLanguageUseTestXiFinder` $= 6 = \overline{110}^2$: runs a XiFinder test, i.e. stores the Fortran Xi's and the C++ Xi's made of Fortran V0's (and stores the Fortran V0's)
- `kLanguageUseTestBothFinders` $= 7 = \overline{111}^2$: runs a test of both finders sequentially, i.e. stores the Fortran Xi's and the C++ Xis made of C++ V0's, as well as the Fortran and C++ V0's.

V0LanguageUsage is a 2-digit binary number. Noting this number xy, digit y tells if the Fortran V0's have to be kept or not, and digit x tells if the C++ V0's have to be found or not. A value of 1 for a considered digit means that the configuration that it represents will be stored in the final file. So here are the possible values and what will be found on the output file :

- `kV0LanguageUseFortran` $= 1 = \overline{01}^2$: only Fortran V0's,
- `kV0LanguageUseCpp` $= 2 = \overline{10}^2$: only C++ V0's,
- `kV0LanguageUseBoth` $= 3 = \overline{11}^2$: both Fortran and C++ V0's.

XiLanguageUsage is pretty similar to V0LanguageUsage, except that it's a 3-digit binary number xyz. z is for the Fortran Xi's, y for the C++ Xi's made of Fortran V0's, and x for the C++ Xi's made of C++ V0's. The rules are the same :

- `kXiLanguageUseFortran` $= 1 = \overline{001}^2$: only Fortran Xi's,
- `kXiLanguageUseCppOnFortranV0` $= 2 = \overline{010}^2$: only C++ Xi's made of Fortran V0's,
- `kXiLanguageUseCppOnCppV0` $= 4 = \overline{100}^2$: only C++ Xi's made of C++ V0's,
- `kXiLanguageUseFortranAndCppOnFortranV0` $= 3 = \overline{011}^2$: Fortran Xi's and C++ Xi's made of Fortran V0's,
- `kXiLanguageUseFortranAndCppOnCppV0` $= 5 = \overline{101}^2$: Fortran Xi's and C++ Xi's made of C++ V0's,
- `kXiLanguageUseBothCpp` $= 6 = \overline{110}^2$: all C++ Xi's (i.e. made of Fortran and or C++ V0's),
- `kXiLanguageUseAll` $= 7 = \overline{111}^2$: all Xi's (Fortran, and all C++ Xi's).

Tables FIG. 13 and 14 are the Karnaugh maps[1] that give the formula giving *V0LanguageUsage* knowing *LanguageUsage* (this is only in the case of LanguageUsage being different than `kLanguageUse-Special` of course). xy are the digits of V0LanguageUsage, abc are those of LanguageUsage.

x		bc			
		00	01	10	11
a	0	X	0	1	X
	1	X	1	0	1

$\overline{a \oplus c}$		bc			
		00	01	10	11
a	0	1	0	1	0
	1	0	1	0	1

FIG. 13 : *Karnaugh map giving digit x of V0LanguageUsage.*

y		bc			
		00	01	10	11
a	0	X	1	0	X
	1	X	1	1	1

$a + c$		bc			
		00	01	10	11
a	0	0	1	0	1
	1	1	1	1	1

FIG. 14 : *Karnaugh map giving digit y of V0LanguageUsage.*

Tables FIG. 15, 16 and 17 are the Karnaugh maps[2] that give the formula giving *XiLanguageUsage* knowing *LanguageUsage* (as for V0LanguageUsage, this is only in the case of LanguageUsage being different than kLanguageUseSpecial). xyz are the digits of XiLanguageUsage, abc are those of LanguageUsage.

x		bc			
		00	01	10	11
a	0	X	0	1	X
	1	X	0	0	1

$b \cdot (\overline{a \oplus c})$		bc			
		00	01	10	11
a	0	0	0	1	0
	1	0	0	0	1

FIG. 15 : *Karnaugh map giving digit x of XiLanguageUsage.*

y		bc			
		00	01	10	11
a	0	X	0	0	X
	1	X	0	1	0

$ab\overline{c}$		bc			
		00	01	10	11
a	0	0	0	0	0
	1	0	0	1	0

FIG. 16 : *Karnaugh map giving digit y of XiLanguageUsage.*

[1]Except that I haven't used the Gray binary code for ab, but the functions are simple here.
[2]Cf. previous footnote.

z		bc			
		00	01	10	11
a	0	X	1	0	X
	1	X	1	1	1

$a+c$		bc			
		00	01	10	11
a	0	0	1	0	1
	1	1	1	1	1

FIG. 17 : *Karnaugh map giving digit z of XiLanguageUsage.*

The formulas can be deduced from those Karnaugh maps :

$$\begin{cases} \text{V0LanguageUsage} = 2(\overline{a \oplus c}) + (a + c) \\ \text{XiLanguageUsage} = 4\,(b(\overline{a \oplus c})) + 2(ab\overline{c}) + (a + c) \end{cases}$$

and their result is shown in the 3 first columns of the tables FIG. 18 and 19, page 32. The first column is LanguageUsage, and the 2 others are V0LanguageUsage and XiLanguageUsage as binary numbers.

As said above, one can chose his own values of V0LanguageUsage and XiLanguageUsage, independantly of *LanguageUsage* (whose value must then be set to `kLanguageUseSpecial` = 0, so as to not overwrite the private choice of V0LanguageUsage and XiLanguageUsage). Not all the combinations are authorised though, because some of them are either meaningless, or too difficult to implement for the use that they would have. A total of 11 combinations are possible.

If all the V0's (i.e. Fortran and C++) are stored, then you can chose any XiLanguageUsage option.

If you don't store the Fortran V0's, then the only XiLanguageUsage option allowed is storing the C++ Xi's made of C++ V0's. Any other value set will cause the code to print a warning message and overwrite the option chosen by `kXiLanguageUseCppOnCppV0`.

If you don't store the C++ V0's, the 3 authorised values for XiLanguageUsage are those which don't require to store the C++ Xi's made of C++ V0's (namely `kXiLanguageUseFortran`, `kXiLanguageUseCppOnFortranV0` and `kXiLanguageUseFortranAndCppOnFortranV0`). Any other choice will lead to a warning printed by the code, and your choice will be overwritten by `kXiLanguageUseFor-tranAndCppOnFortranV0`.

An important thing to mention is that the V0's and Xi's found by the C++ codes are flagged, so as to differentiate them from the Fortran candidates that may be kept. Any V0 candidate found by the (C++) V0Finder has a negative χ^2, and any Xi candidate found by the (C++) XiFinder has a negative χ^2, except for those which are made of a Fortran V0 (since code version 1.8 of `StXiFinderMaker.cxx`).

Let's see now the effect of these options on container keeping/erasing. When only the V0Finder is run, it's quite simple :

- If the second digit (starting from the least significant bit) of V0LanguageUsage is 0, the V0Finder is not run at all. As an effect, not C++ V0's are found and the Fortran V0's are kept.
- If the first digit of V0LanguageUsage is 0, the containers of Fortran V0's and Xi's are erased, and then the V0Finder is run.

For the XiFinder, things are a bit more complicated... Let's call XiLanguageUsage xyz, and V0LanguageUsage $x_v y_v$: if $z = 0$, the initial container of Xis is erased (because $z = 0$ means that we don't want to keep the Fortran Xis). Then, if $y = 1$, a loop over the existing V0's is done and method `UseV0` (i.e. the XiFinder algorithm) is called for each of them. This is how the C++ Xi's made of Fortran V0's (the pre-existing ones) are made. And finally, if $x = 1$ and/or if $x_v = 1$, the V0Finder is called (the first case means we want C++ Xi's made of C++ V0's – so C++ V0's have first to be found –, the second one means that we want C++ V0's).

The existing container of Xi's is actually not removed (even though $z = 0$) in one case : when $x = 1$ and $y_v = 0$. The only case when this actually happens is for (V0LanguageUsage,XiLanguageUsage)

= (10,100), and the reason for that is that in such a case, because $x = 1$, the V0Finder is called and the container that we want to remove *is* removed in the V0Finder, as soon as $y_v = 0$. The container can't be removed twice, so it's removal is prevented in the XiFinder by requiring the condition $x = 1$ and $y_v = 0$. In the theory, we should also require – in order to remove the Xi container – that "$x_v = 1$ and $y_v = 0$ is false", because a call to the V0Finder is also made when $x_v = 1$. Yet, this is not necessary in practice, because the only possible case "$x_v = 1$ and $y_v = 0$" is (V0LanguageUsage,XiLanguageUsage) = (10,100), that is already taken care of.

Then, at the beginning of function `UseV0` of the XiFinder, a test whether C++ Xi's are required to be found is done : if $x = 0$ and $y = 0$, it means that no C++ Xi's are asked, and we exit of the function with return code *false*, meaning that the V0 hasn't been used to make a Xi.

The last test that is performed is also at the beginning of function `UseV0` of the XiFinder on concerns the difference between C++ Xi's made of Fortran V0's and C++ Xi's made of C++ V0's. Since we want both of them when XiLanguageUsage equals 6 or 7 (respectively kXiLanguageUseBothCpp and kXiLanguageUseAll), this test is run only for XiLanguageUsage < 6. It just exits from the function (also with exit code *false*) when the χ^2 of the V0 is negative if $y = 1$ (the V0 was a C++ one), and when the χ^2 of the V0 is positive or null if $x = 1$ (the V0 was a Fortran one).

I've explained this in a Fortran vs C++ framework for historical reasons, but of course one can use these options to play with container keeping/erasing/filling for other purposes, i.e. not necessarily Fortran candidates may be present in the pre-existing containers : these containers may be empty, or may be filled with C++ candidates coming from a previous pass.

Options			Maker is run		Pre-existing V0's		Pre-existing Xi's	
	V0	Xi	V0Finder	XiFinder	Erased	Kept	Erased	Kept
1	01	001				X		X
	01	010		X		X	X	
6	01	011		X		X		X
2	10	100	X	X	X		X	
5	11	001	X			X		X
	11	010	X	X		X	X	
	11	011	X	X		X		X
	11	100	X	X		X	X	
7	11	101	X	X		X		X
	11	110	X	X		X	X	
	11	111	X	X		X		X

FIG. 18 : *Effect of the available options on the makers and pre-existing candidates.*

The table FIG. 18 sums up which algorithms are run / not run, and which containers are kept / erased depending on the option choice. The first column gives the LanguageUsage option that sets the equivalent pair of options (V0LanguageUsage,XiLanguageUsage). The next table, FIG. 19, translates this in terms of what happens to the V0 and Xi containers : you can see whether they are erased (it actually never happens), kept as is, appended or overwritten in function of the options set, or you can on the contrary decide which options you have to use depending on the actions you want to be taken on the containers.

Options			V0 container				Xi container			
	V0	Xi	Erase	Keep	Append	Overwrite	Erase	Keep	Append	Overwrite
1	01	001		X				X		
	01	010		X						X
6	01	011		X					X	
2	10	100				X				X
5	11	001			X			X		
	11	010			X					X
	11	011			X				X	
	11	100			X					X
7	11	101			X				X	
	11	110			X					X
	11	111			X				X	

FIG. 19 : *Effect of the available options on the V0 and Xi containers.*

8.4 Beware of the flags !

A couple of flags are set by the V0/XiFinders when finding the candidates. Those flags are melted in the candidates' variables. There are 4 flags :

- Tracker flag : ITTF vs TPT,
- Finder flag : Fortran vs C++,
- SVT flags,
- V0 flag : primary vs secondary V0.

ITTF flag : for TPT candidates, the distance of closest approach between the V0 daughters is positive. For the ITTF candidates, it's negative. This encoding is the same for the V0's and for the Xi's, because an ITTF V0 can be combined only with an ITTF track to form a Xi, so a Xi is also flagged by the sign of the dca between the *V0* daughters.

Finder flag : it differentiates the Fortran and C++ V0/Xi candidates. The Fortran candidates have a positive or null χ^2 (χ^2_{V0} for the V0, χ^2_{Xi} for the Xi), while the χ^2 of the C++ candidates is set to a negative value in the V0Finder (χ^2_{V0}) and XiFinder (χ^2_{Xi}). Be careful : the Fortran's χ^2's are often *equal* to zero, rather than strictly positive. An important additionnal note : since version 1.8, the XiFinder does NOT set a negative χ^2_{Xi} when it's making Xi's out of Fortran V0's : the χ^2_{Xi} can have any value, positive as well as negative (it actually depends on the SVT usage for the considered condidate).

SVT flag : to be written.

V0 flag : if the V0 is primary, the distance of closest approach of the V0 trajectory to the primary vertex (impact parameter) is positive. If the V0 is not primary, then the dca between the V0 and the primary vertex is negative. Notice that a secondary V0 could also be classified as primary ; the condition for the flag to be set is not that the V0 is secondary, but that it is not primary. So the set of the flagged V0's is only a subset of the secondary V0's.

9 Tests

I have unfortunately no time to write this section in details. There are basically 3 types of tests : whether the code runs and does find Xis, whether the variables inside the XiFinder have the same value in C++

and in Fortran, and whether the number of Xis found and their distributions are similar.

In this paragraph, I'll talk mainly about the XiFinder. The XiFinder algorithm is exactly the same as the Fortran one, the V0Finder one is not.

The version of the code that has to be used for comparisons with the Fortran code is :

- `StV0FinderMaker.cxx` : version 1.17 without the causality cut introduced in version 1.9[1] (last version for now is 1.18)
- `StV0FinderMaker.h` : version 1.7 (the last one for now)
- `StXiFinderMaker.cxx` : version 1.18 (last one for now is 1.19)
- `StXiFinderMaker.h` : version 1.2 (the last one for now)

In more recent versions of the Finders, new cuts have been introduced compared to Fortran (see § 4.3 p. 10 and § 4.4.4 p. 22 for more information).

9.1 Invariant mass peak

The code does run, in the production as well as stand-alone. After filtering the muDsts produced with tight cuts, an invariant mass peak appears, that can be compared with the Fortran invariant mass peak. This is shown in FIG. 20.

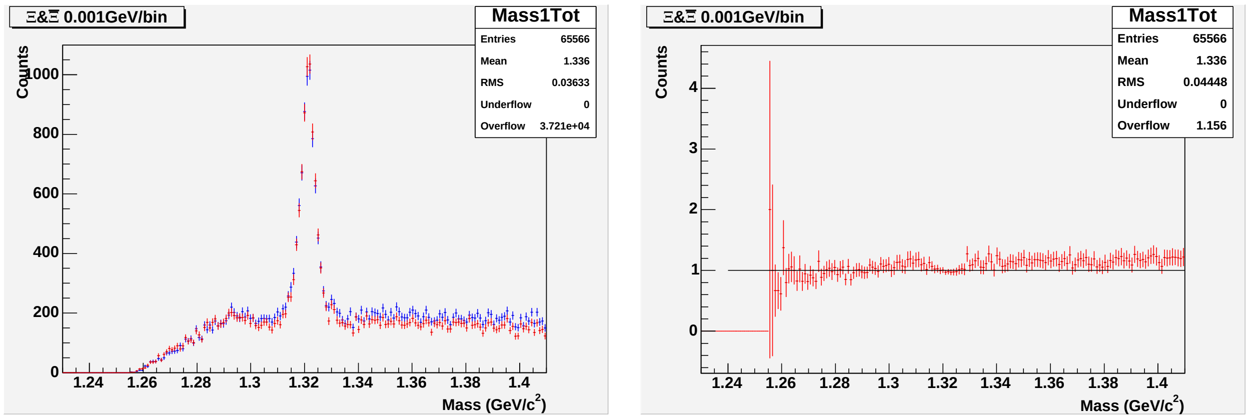


FIG. 20 : *Left plot : comparison of the XiFinders : Fortran invariant mass distribution in blue, C++ in red. Right plot : ratio Fortran/C++.*

Details can be found here :
`www.star.bnl.gov/protected/strange/faivre/utilities/xifindTest.html`

9.2 Deep level tests

Unlike the 2 other series, the first series of tests that I've made have been done only on the XiFinder (although I've made a V0Tester, but only to test the final variables and which V0's were found or not – since the algorithm is different, deep level tests can't be made on the V0Finder). Furthermore, they test essentially... the background and not the signal, because of the so small number of signal Xi's ! (And technically : it was run in the BFC, so was very time- and disk-space-consuming, so only a few events could be processed). It consisted in printing the value of several or all the variables in the XiFinder algorithm, in Fortran and in C++, and comparing those values on a candidate-per-candidate basis.

[1]This cut is labelled "Cut: check if the first point of either track is after v0vertex".

Roughly speaking, I've printed 1/10th of the final results, and those forms a 3-cm high stack of paper sheets... So giving all the details is out of question ;-)

Since the Fortran maker and the C++ maker are run one after the other, the Xi vertices have to be identified. The event is identified by the position of the primary vertex, then the V0 is identified by the position of its vertex, and then the bachelor is identified with its momentum. This makes a unique Xi in Fortran and in C++. The codes that I've called "testers" basically associate a Fortran Xi with the corresponding C++ Xi (if it exists), and then plot some distributions showing the differences. More details can be found in the talk that I gave during the strangeness workshop of November 2002 in UCLA, which can be found here :
`www.star.bnl.gov/protected/strange/faivre/talks/20021018/tlk2j2.html`

I've also inserted bugs on purpose and checked what was the effect on the output plots I was looking at : the effect has been in all 4 cases immediately visible.

According to the type of the variables used, the error due to digitalisation and binary rounding of the floating-point numbers that we have to expect on the variables is :

$$6.10^{-8} < \frac{\Delta x}{x} < 1.2\ 10^{-7}$$

All 304 XiFinder variables have been checked, and the results are definitely satisfactory. The (very) few outliers were due to no value printed in the logfile, either in Fortran or in C++, for various reasons ; so they are not caused by a bug in the code.

9.3 Global tests

This series of tests consists in forgetting about the candidate-by-candidate comparisons, and checking that the distributions of various XiVertex variables ae similar between Fortran and C++, plus checking that the number of candidates found is the same. The tests are described here :
`www.star.bnl.gov/protected/strange/faivre/utilities/xifindTest.html`
and all the plots of the distributions can be seen on the same webpage.

Those tests show that the distributions and number of candidates found by the Fortran XiFinder and the C++ XiFinder are exactly similar when analysis cuts are applied, and almost exactly similar (see the webpage for details) when no cuts are applied : the difference can't be seen with the eye, because it's almost always below 1 σ of the statistical error bar.

Tests of the V0Finder show that some distributions are significantly different, but since the algorithm is different it's expected. The important thing is that when the invariant mass plots are compared once the analysis cuts are applied, no difference is seen in the number of signal found in the peak.

10 To-do list

There is no particular order and not everything is relevant/worth doing ;-)

- fabs vs TMath::Abs.
- TThreeVector vs double[3].
- Write more about tests in the documentation.
- During the tests, I've seen that some tracks had all their parameters equal to 0 ; I've set something in the code to prevent this (track is skipped, it avoids code crashing for dividing by 0 when inverting the curvature, for example), but haven't tried to understand why those tracks existed.
- Fortran code doesn't treat correctly the cascade candidate when its V0 mass falls both in the Λ and in the $\overline{\Lambda}$ mass hypothesis (it does only one of the 2 hypothesis). Influence on the tests ? (Concerns only a very small percentage of the candidates).
- Bfield value is calculated from the momentum of the 1st track of the container, whatever its $p_\perp$: could be useful to calculate Bfield only with a track that has a curvature reasonably high (i.e. not too high $p_\perp$).
- Remove usage of `xAns` and `yAns` in `StXiFinderMaker.cxx` : use directly `xOut` and `yOut`.
- Flag χ^2_{Xi} negative even when the V0 is a Fortran one.
- Change XiFinder algo : allow at most 1 stored Xi candidate per pair (V0,track). To choose : take the intersection point that gives the smaller dca.
- Etc...